Using GPUs for High Performance Computing Applications

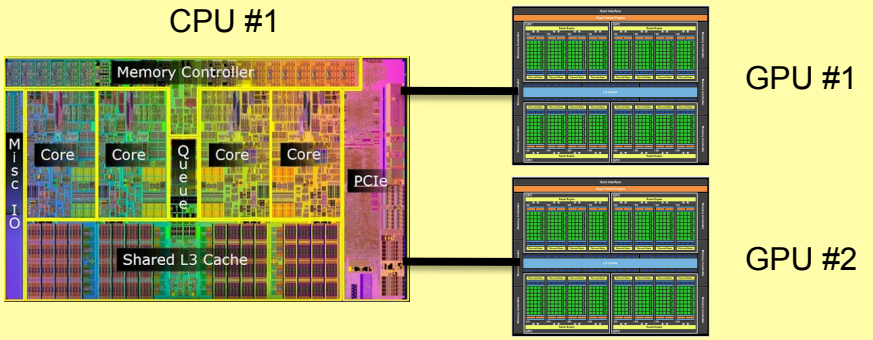
Mark Govett
Craig Tierney
Jacques Middlecoff

Outline

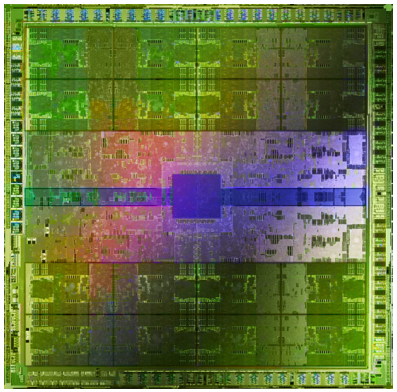
- GPU Basics
- NOAA Research Efforts
  - F2C-ACC Compiler Development
  - NIM Parallelization and Performance

NVIDIA GPUs

Illustration of two Fermi GPUs attached to a dual-socket Nahalem CPU



Fermi (2010)



✧ 8x increase in double precision
✧ 2x increase in memory bandwidth
✧ Error correcting memory

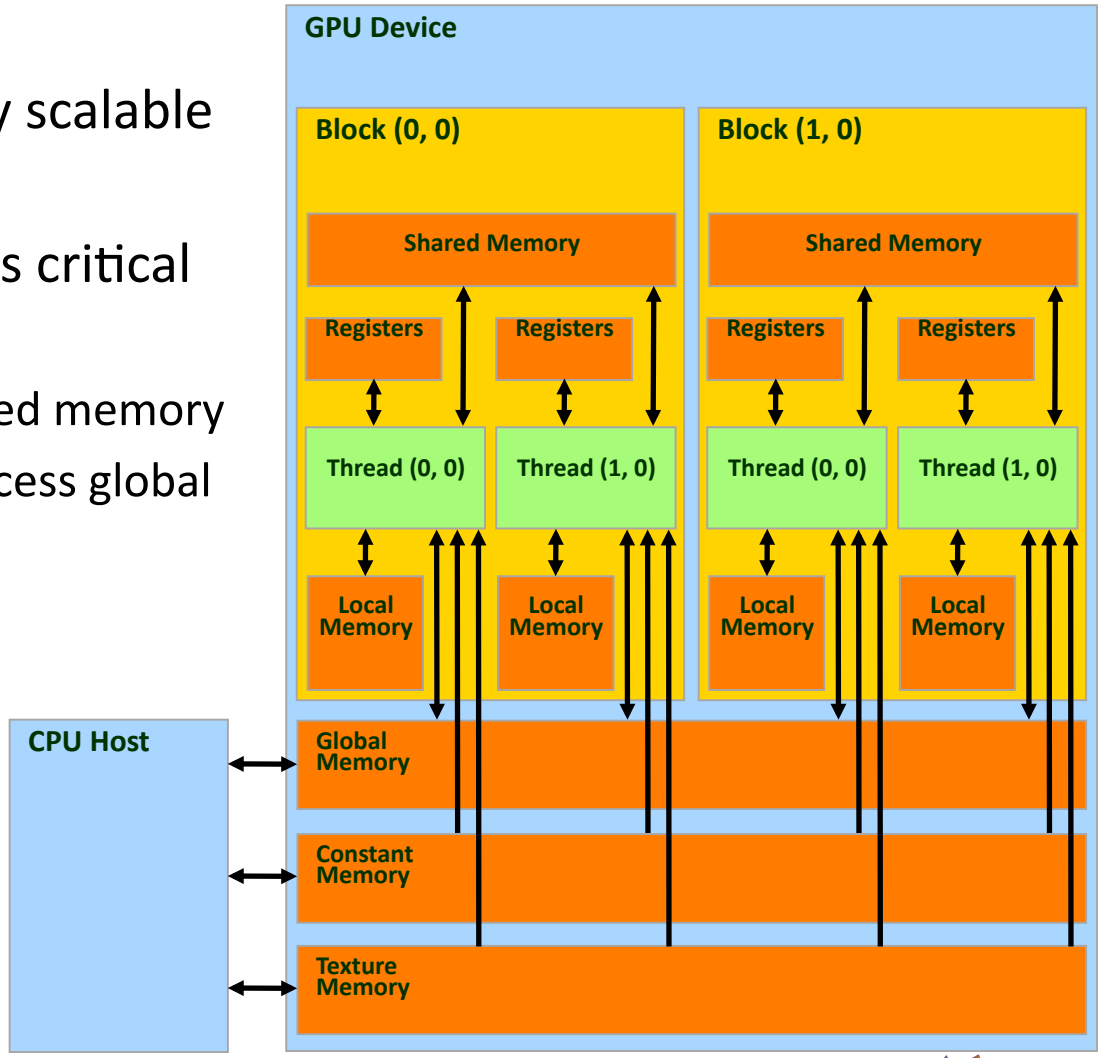
Tesla (2008)

GPU: 2008
933Gflops
150W

CPU:2008
~45 Gflops
160W

GPU Memory Model

- 100x is possible on highly scalable codes
- Efficient use of memory is critical to good performance
  - 1-2 cycles to access shared memory
  - Hundreds of cycles to access global memory

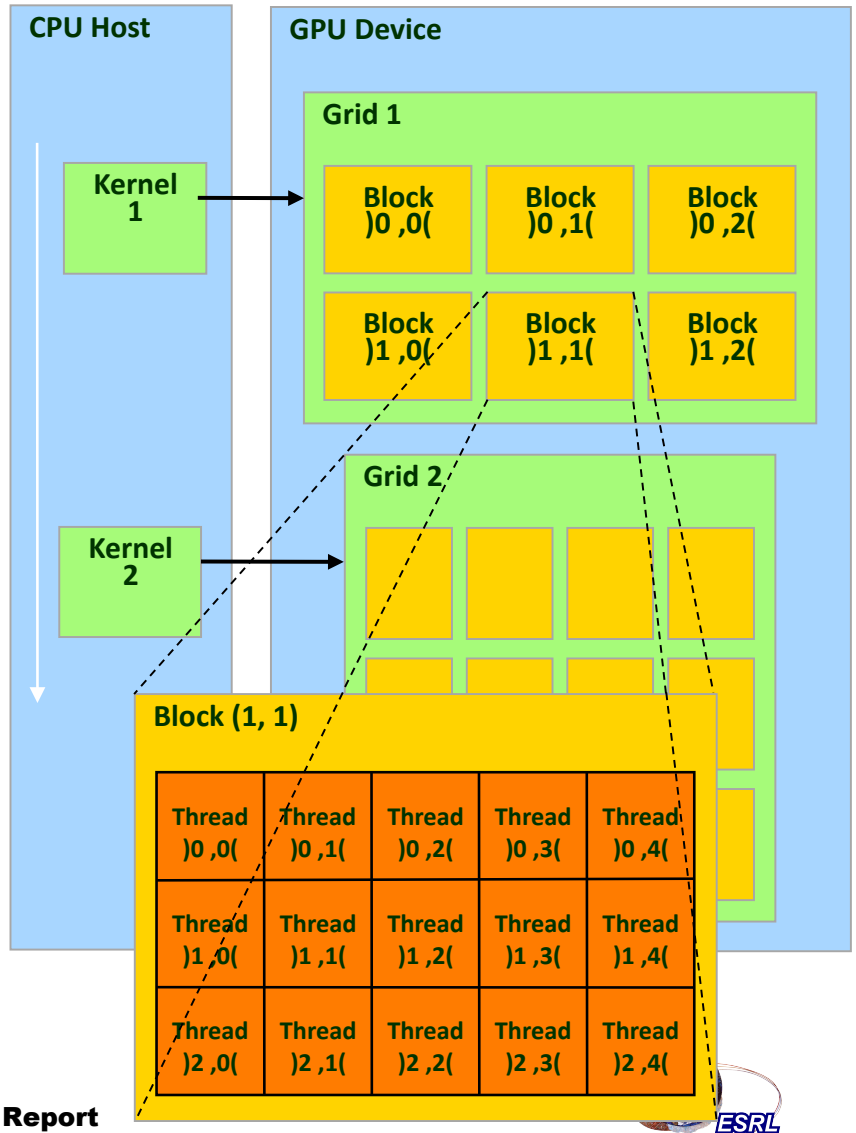


Tesla (2008)

- 16K shared memory
- 16K constant memory
- 2GB global memory

GPU Programming Model

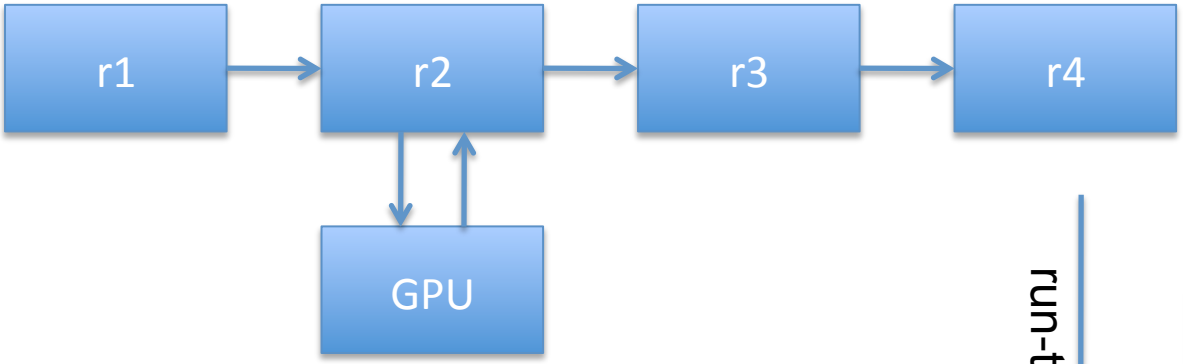
- Kernels – program segments
- Thread Blocks - blockId
- Threads – threadID

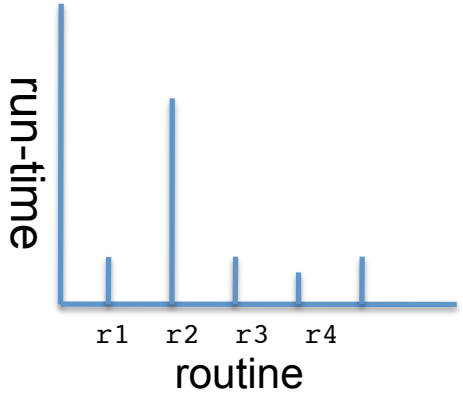


Tesla (2008)

- 30 Multiprocessors
  - 1024 "active" threads / multiprocessor
    - 8 blocks
      - 32 threads execute in lockstep
- Resource Limits
  - 16384 registers / block
  - 16K shared memory / block
  - Maximum 512 threads / block

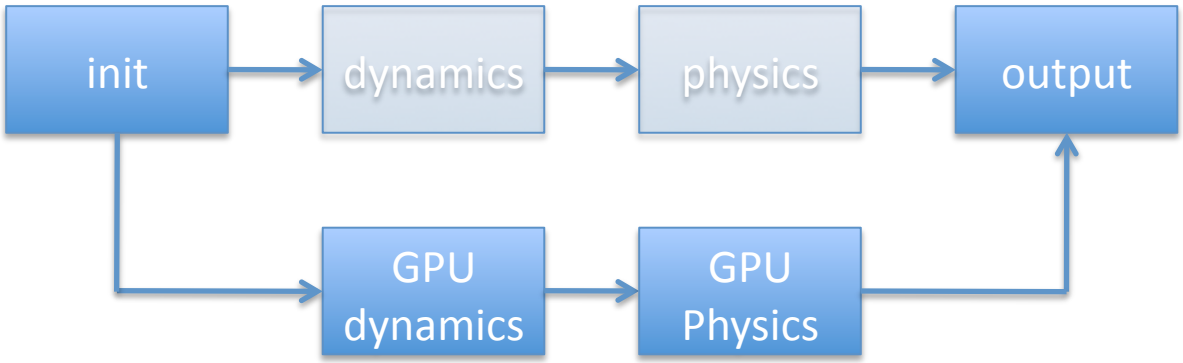
Execution Flow-control
(select routines)





– Copy between CPU and GPU is non-trivial
  - Benefits can be overshadowed by the copy
  - WRF model demonstrated 7x improvement (including copies)

Execution Flow-control
(run mostly on the GPU)



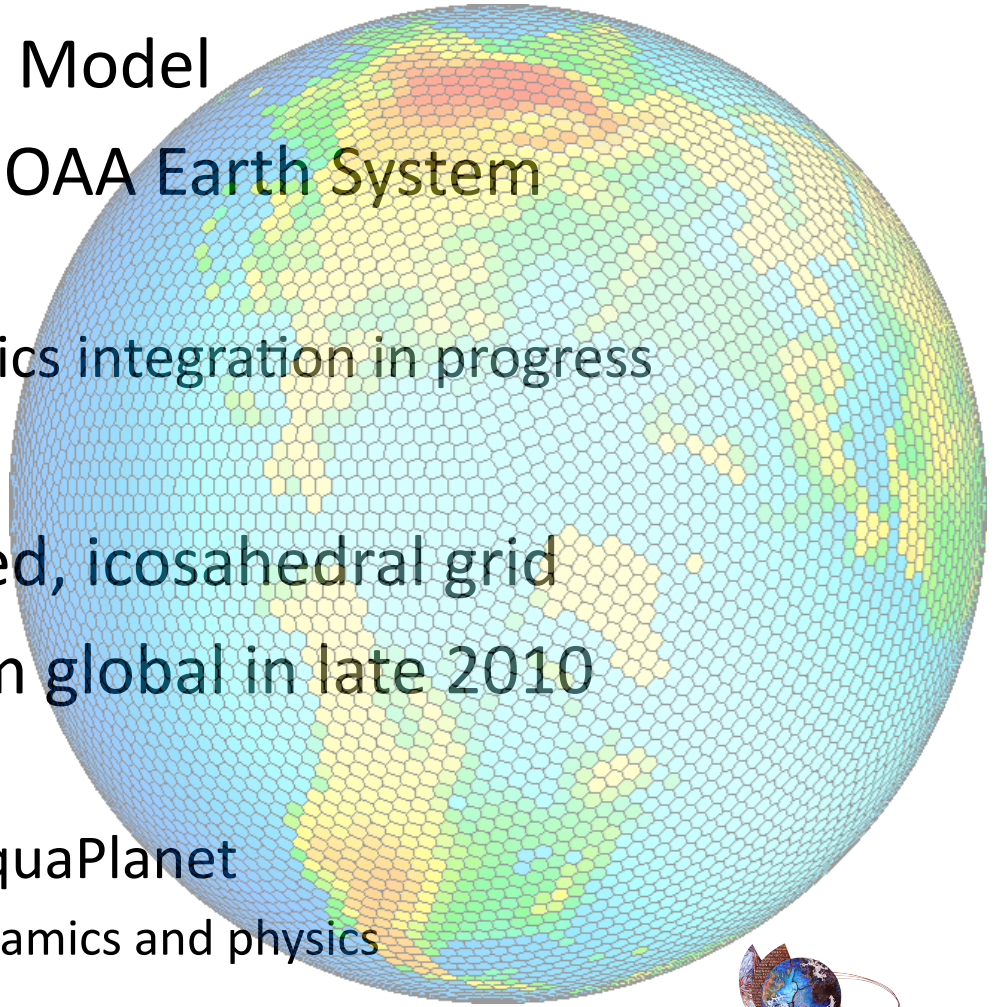
- Eliminates copy every model time step
- CPU-GPU copy only needed for input and output

ESRL Research Efforts

- Next Generation Weather Models are driving computing requirements
  - NIM Model development (MacDonald, Lee)
- Purchased 16 node NVIDIA Tesla system in 2008 (~31Tflops)
- Developed Fortran to CUDA compiler
- Parallelized NIM model dynamics

Non-hydrostatic Icosahedral Model (NIM)

- Global Weather Forecast Model
- Under development at NOAA Earth System Research Laboratory
  - Dynamics complete, physics integration in progress
- Non-hydrostatic
- Uniform, hexagonal-based, icosahedral grid
- Plan to run tests at 3.5km global in late 2010
  - Cloud resolving scale
  - Model validation using AquaPlanet
    - Idealized test for both dynamics and physics



Next Generation Weather Models

- Models being designed for global cloud resolving scales (3-4km)
- Large CPU systems (~200 thousand cores) are unrealistic for operational weather forecasting
    - Require forecasts to be generated at 2 percent of real-time
    - Poor application scaling
    - Power, cooling, reliability, cost

<u>GPU System</u>

- 1.0 PetaFlop
- 1000 NVIDIA GPUs
- 10 cabinets
- 0.5 MW power
- ~ $5 million
- MTBF in weeks

<u>DOE Jaguar System</u>

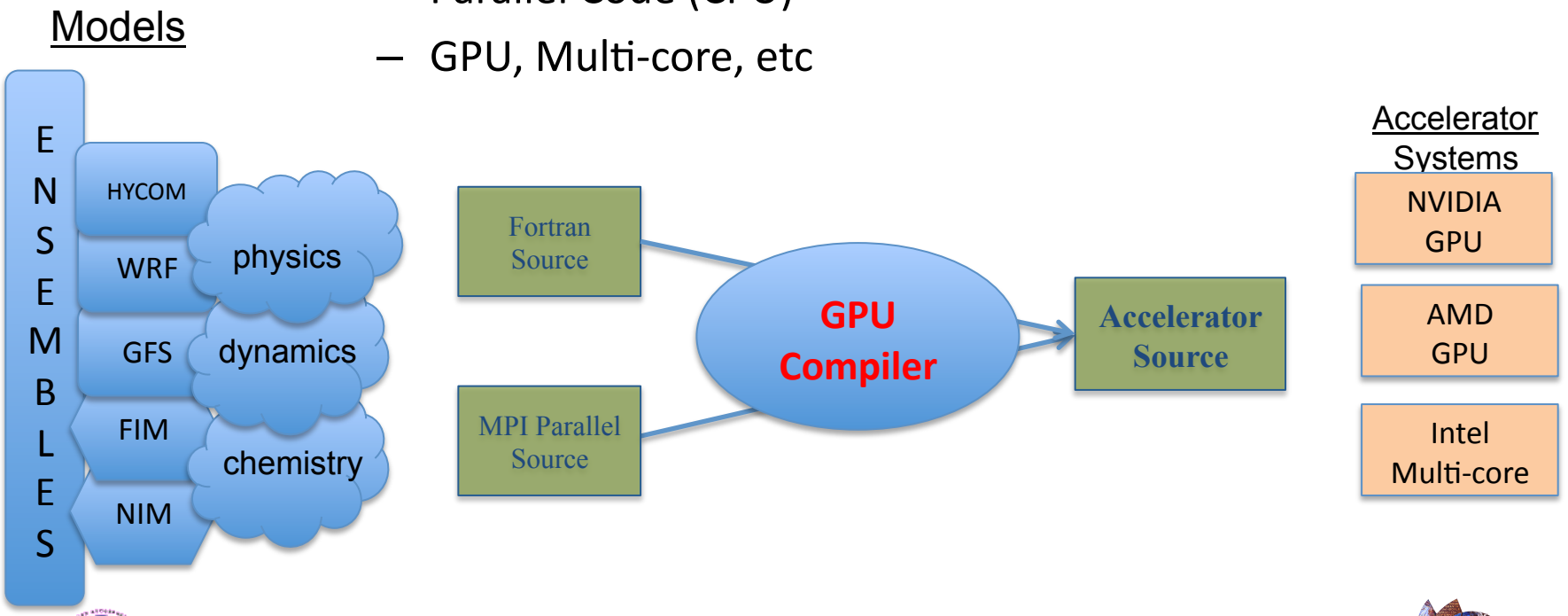
- 2.3 PetaFlops
- 250,000 CPUs
- 284 cabinets
- 7-10 MW power
- ~ $50-100 million
- MTBF in hours

Fortran-to-CUDA Translator

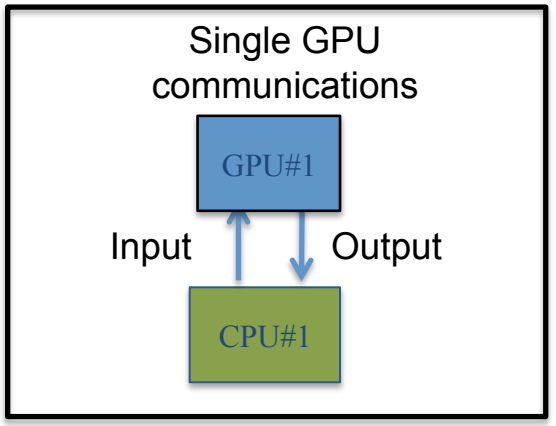
- Motivation
  - In 2008 it was unclear if/when NVIDIA would support Fortran or what the capabilities will be
  - Hand translation is too slow
  - Maintain a single source code
- Converts Fortran 90 into C or CUDA-C
  - Some hand tuning is necessary
- Future plans dependent on vendor solutions

Parallelization & Portability

- Directive-based GPU Parallelization
- Single Source Code
  - Serial Code (CPU)
  - Parallel Code (CPU)
  - GPU, Multi-core, etc



Status of FY09 Work

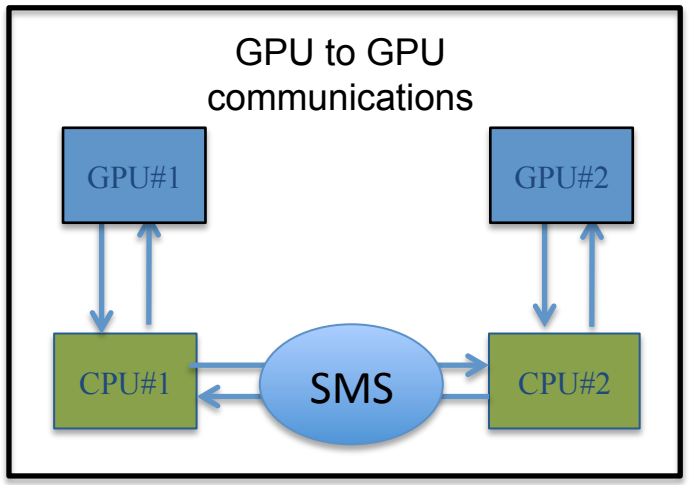


- Dynamics
  - Horizontal data dependencies
    - GPU threads over 96 vertical levels
  - Communications only needed for I/O
  - Limited by 1GB GPU memory
    - Increased number of blocks hides memory latency
  - Runs 34 times faster on a GPU than CPU
- Physics
  - WRF microphysics being incorporated
  - Dominated by vertical dependencies

Using GPUs for High Performance Computing Applications – Part II

(the HPCC FY 2010 Proposal)

- Run NIM on Multiple GPUs
  - Modifications to SMS libraries to pack and unpack data to be communicated via MPI
  - Upgrades F2C-ACC compiler
    - Data movement
    - Distributed memory addressing
    - Data persistence
- Evaluate Fortran GPU compilers
  - Maintain codes in Fortran
    - Not practical to translate into CUDA

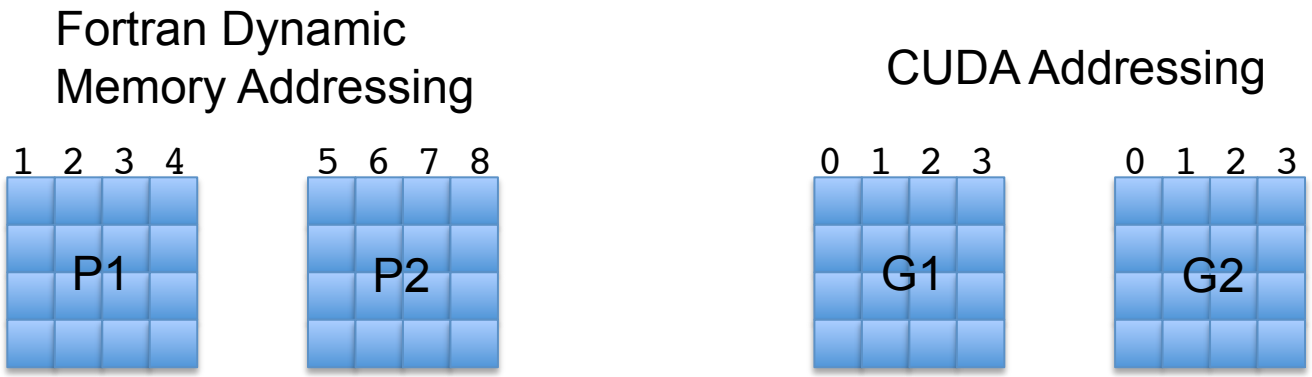


Memory Addressing

- C / CUDA require arrays start at 0
- Array references are collapsed using macros

```
a(i,j) becomes    a[FTNREF2D(i,j,nx,1,1)]
```

- Multiple GPUs addressing becomes complex
  - `a[FTNREF2D(i,j,nx/2,lbound,ubound)], where`
    `Lbound, ubound are calculated upper and lower bounds`
- Conversion of loops, arrays to local address



Data Persistance

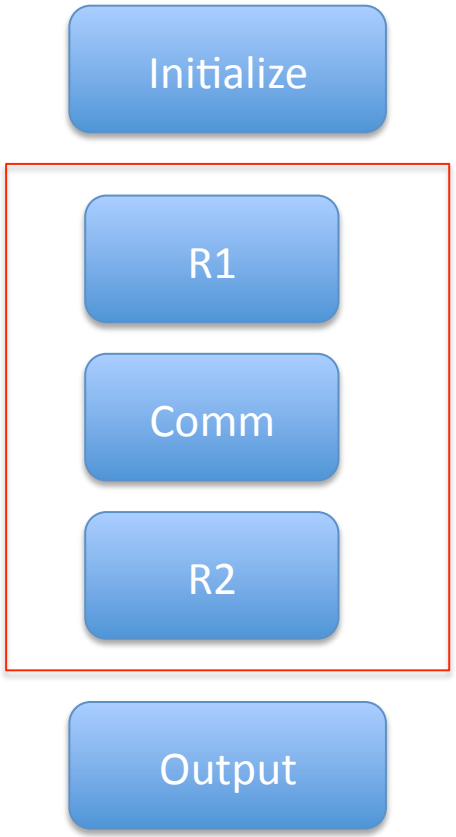
- Use cudaMalloc and cudaMemcpy to allocate arrays on GPU during initialization

```
float *d_ur;
cudaMalloc((void **) &d_ur,((nz+1))*sizeof(float));
cudaMemcpy(d_ur,ur,(nz+1)*sizeof
                (float),cudaMemcpyHostToDevice);
```

- Pass the pointer to the data thru the argument list

```
kernel<< cuda_grids,cuda_threads >>>(d_ur, …>>>
```

- Use cudaMemcpy to copy data back to the CPU
  - Only copy halo when doing inter-process communications
  - Only copy entire array for model output



Conclusion

- GPUs are an attractive HPC platform for scientific computing with tremendous potential
  - We demonstrated 34x performance boost
- Compilers need to mature to simplify parallelization, conversion
  - Users still need to analyze, convert their codes
- F2C-ACC will be used to support parallelization
  - Rely on PGI, CAPS where possible